Today's plan

- Minix scheduling
- Interprocess communication: pipes
- Message passing, including in Minix
- Monitors
- Minix context switch
- Minix interrupt handling

Pipes

- ideal for consumer-producer problem
- on Unix/Posix: pipes of bytes. In theory pipes of arbitrary data structures could also be useful
- on Unix/Posix can use arbitrary file operations, including read and write, select, and close.
- consumer reads from pipe, up to a maximum specified by the read call, may block if no data is available
- producer writes to pipe, may block if no buffer space is available

Implementation of Pipes

- in a system with very little memory, could synchronize producer and consumer to copy data from producer's buffer directly to consumer's buffer
- in practice a system buffer gives better performance than no buffer, so most systems allow the producer to write up to a certain amount of data (typically 4096 bytes), which consumer can read all at once
- a single thread that is both producer and consumer of the same pipe will block if the process tries to write more than the buffer holds

Message Passing

- can extend pipes to pass structured data rather than just bytes
- can also extend pipes to data among different processors or across a network
- this is called **message passing**
- message passing systems must answer questions of:
  - identifying receiver, e.g. a process or a mailbox (a mailbox could be accessed via a file descriptor, e.g. a socket)
  - dealing with lost messages or overflowing buffers: acknowledgements, sequence numbers
  - efficiency, avoiding copying unless absolutely necessary -- important on message-passing supercomputers

Synchronization for Message Passing

- message passing may not provide enough synchronization, so external forms of synchronization, e.g. barrier synchronization, can be used
- another form of synchronization is to have the sender **rendezvouz** (first process to reach the rendezvouz point waits for the other) with the receiver, so data can be copied directly from the sender's to the receiver's buffer
- much in common with networking, but sometimes on a single system

Minix Message Passing

- sender has a message in a buffer that it wants to give to a specific receiver
- receiver has a message buffer that it wants to fill from a given or from any sender
- first one to send or receive blocks
- blocked sender is queued on receiver's process table entry
- all "regular" (Posix) system calls are implemented by sending a message to the File System or the Process Manager server (or the INET server)
- all system calls send, then receive (sendrec), otherwise a caller could block a server forever
- message passing also used among tasks and the kernel

Monitors

- a monitor is a module (similar to a C++/Java class) with some private variables that only the monitor routines can access
- at most one monitor routine can be active at any given time
- when a second monitor routine is called, it suspends (before executing) until the first has completed
- a monitor routine never needs to be re-entrant, that is, it is guaranteed to complete before the next call to the same routine (or any other routine in the same monitor)
- Java `synchronized` classes are similar to monitors
- a monitor routine that cannot continue can wait, and another is allowed to run (and eventually signal the one that called wait)
- writing re-entrant code is hard, and errors are hard to debug, so monitors help write correct concurrent programs

Common Practice

- monitors require language support
  - e.g. Java synchronized classes
- without language support, programmers must
  - identify related global variables that are changed as individual critical regions
  - create a lock associated with each group of variables
  - acquire the lock each time before reading or modifying the variables
  - release the lock each time after reading or modifying the variables

```
lock.acquire ()
```

critical region

```
lock.release ()
```

- these operations are error-prone
- failure to release a lock may deadlock the entire system, since all processes may eventually be waiting for the lock to be released
- it is easy to forget to release a lock -- just return from within the critical region

Minix context switch

- context switch on (hardware) interrupt or on system call (software interrupt)
- interrupt from user mode will reload stack pointer from a given location in memory (book, p. 168-169)
- Minix updates this location every time a new process is started so registers are automatically saved in the process descriptor, the process table entry for the currently running process
- some registers are saved automatically, and others are pushed by save, which also increments _k_reenter (p. 712)
- the result is in sigregs (p 658) format
  - newer version: trapframe/intrframe (https://github.com/Stichting-MINIX-Research-Foundation/minix/blob/master/sys/arch/i386/include/frame.h)

Interrupts in Minix

- see comments in kernel/mpx386.s, p. 707
  - `minix/kernel/arch/i386/mpx.S` at https://github.com/Stichting-MINIX-Research-Foundation/minix/
- hardware interrupt causes execution of `hwint00..hwint15`
- these routines call `save` (p. 712, or `SAVE_PROCESS_CTX` in `minix/kernel/arch/i386/sconst.h`), which saves the register, sets up a kernel stack if needed (if `_k_reenter` >= 0), adds a return address that calls `_restart`, and returns
- these routines then call `intr_handle` (kernel/i8259.c, p. 735), which is written in C, with an argument to tell it which interrupt it is handling (or call `irq_handle` in `minix/kernel/interrupt.c`, then `switch_to_user` in `minix/kernel/proc.c`)
- once `intr_handle` returns, these routines disable the specific interrupt (should be re-enabled by the device driver), re-enable interrupts in general, and return
- the return goes to the address set up by `save`, which will either return to the kernel code that was interrupted, or start or restart a process, often not the one that was interrupted -- for example, a newly awakened task, specified by next_ptr in restart (p. 713)
  - in the current version, `irq_handle` transfers control to `switch_to_user`
- this new task will typically be a device driver

Device-Specific Interrupt Handling and `intr_handle`

- `intr_handle` is given a list of hooks (function pointers), stored in the `irq_handlers` array, and calls them in turn
- `irq_handlers` are set up by calls to `put_irq_handler` which, except for the clock device, is called by `do_irqctl` in kernel/system/do_irqctl.c (not in book)
- `do_irqctl` calls `generic_handler`, which transforms interrupts into messages by calling `lock_notify` in kernel/proc.c
- `generic_handler` then conditionally re-enables the specific interrupt by returning a bit
  - in the current version, interrupts are re-enabled as long as none of the hooks returns false
- `lock_notify` calls `mini_notify` after "locking" (disabling interrupts) but only if we are not already re-entering (`lock` is declared on line 4861 of kernel/const.h, p. 692
- `mini_notify` either creates the message and makes the receiver ready to execute, or saves the message information (one bit) if the receiver is already active
- in any case, `intr_handle` never blocks -- if other kernel code is executing, it simply returns after recording that the interrupt was held back